TEMA 2

Tipos Abstractos de Datos.

2.1. CONCEPTO

Como se vio en el apartado 1.4, se entiende por Estructura de Datos una agrupación de datos, simples o compuestos, del mismo o diferente tipo, que constituyen una entidad en su conjunto (por ejemplo un vector o un registro).

Evolucionando en el concepto de estructura de datos aparece el de Tipo Abstracto de Datos (TAD) que obedece a la técnica de abstracción de datos y sienta las bases para una adecuada compresión de la Programación Orientada a Objetos.

Una definición de TAD u Objeto sería: *El conjunto constituido por la estructura de datos y las operaciones asociadas a la misma que permite modelar el comportamiento de una entidad real*.

Se trata de construir entidades (TAD's, Objetos, etc.) que puedan ser posteriormente utilizadas por otros.

Sus principales características son:

- **Ocultamiento.** El TAD tiene un comportamiento de "caja negra" dado que quien lo usa sabe **qué** puede hacer pero no **cómo** lo hace.

- **Encapsulamiento** (y, en consecuencia, **Protección**). El usuario de TAD's no tiene acceso y, por tanto, no puede modificar sus características. No obstante, puede partir de él para construir otros TAD's[1]

- **Compilación separada**: El resultado de la compilación del TAD se pone a disposición de los usuarios en forma de unidades que pueden utilizarse como si estuvieran predefinidas en el lenguaje de programación.

El objetivo de emplear este tipo de mecanismos obedece a criterios de productividad en la Ingeniería del *Software*. De forma simplificada se trata de clasificar a los profesionales de la construcción de *Software* en dos categorías:

- Constructores (y distribuidores) de TAD's.

- Utilizadores de TAD's hechos por otros.

El usuario de un TAD sabe de su comportamiento a partir de las especificaciones tanto semánticas como sintácticas que le habrán sido proporcionadas por el creador del mismo.

Veamos, a partir de unos sencillos ejemplos, la forma de usar TAD's (queda pendiente cómo construirlos: se verá en los temas posteriores). Se trata de dos tipos abstractos de datos: Pilas y Colas que constituyen fenómenos observables con mucha frecuencia en el mundo real.

[1] Esta idea permite introducir el segundo concepto más representativo de la Programación Orientada a Objetos: la **HERENCIA**

2.2. TAD PILA DE NÚMEROS ENTEROS.

2.2.1. Concepto.

Una pila es una agrupación de elementos de determinada naturaleza o tipo (datos de personas, números, procesos informáticos, automóviles, etc.) entre los que existe definida una relación de orden (**estructura de datos**). En función del tiempo, algunos elementos de dicha naturaleza pueden llegar a la pila o salir de ella (**operaciones / acciones**). En consecuencia el estado de la pila varía.

Una pila presenta el comportamiento LIFO (*Last Input First Output*) y el criterio de ordenación se establece en sentido inverso al orden de llegada. Así pues, el último elemento que llegó al conjunto será el primero en salir del mismo, y así sucesivamente.

2.2.2. Modelo gráfico

Podríamos representar gráficamente una pila según aparece en la figura 2.1: una estructura de datos vertical, en la que los elementos se insertan y extraen por la parte superior.

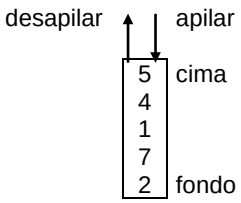

Figura 2.1. Modelo gráfico de Pila

2.2.3. Especificaciones

A continuación se describen las operaciones que vamos a poder realizar sobre el TAD pila de enteros. Para ello, utilizaremos dos tipos de especificaciones:

- Especificaciones sintácticas. Hacen referencia a los aspectos sintácticos de las distintas operaciones asociadas al TAD: nombre de la operación, parámetros, resultado devuelto por la operación, tipo de dicho resultado, etc.
- Especificaciones semánticas. Indican el efecto producido por la operación, es decir, la definen desde el punto de vista de lo que hace.

En la Tabla 2.1., se definen las especificaciones semánticas y sintácticas del TAD pila de enteros. Cuando se desee utilizar el tad pila en un programa, habrá que poner la siguiente sentencia en la primera línea del programa, después de la cabecera (o bien utilizarlo dentro del mismo paquete –*package tadPila*–):

```
import tadPila.*;
```

Operación	Especificación semántica	Especificación sintáctica
apilar [2]	Método que entrega un elemento (x) para que quede incorporado en la cima de la **pila**.	void apilar (int x)
desapilar [3]	Método que elimina el elemento que ocupa la cima de la **pila** y devuelve como resultado dicho elemento.	int desapilar () throws PilaVacia
pilaVacia	Método que al ejecutarse devuelve true si la **pila** está vacía (no tiene elementos), y false en caso contrario.	boolean pilaVacia ()
leerPila	Método que se utiliza para realizar la carga inicial de elementos de la p**ila**.	void leerPila () throws NumberFormatException, IOException
imprimirPila	Método que muestra en la pantalla el contenido de la **pila**.	void imprimirPila ()
numElemPila	Método que devuelve el número de elementos de la p**ila**.	int numElemPila ()
cima	Método que devuelve la cima de la **pila** (sin alterarla).	int cima () throws PilaVacia
decapitar	Método que elimina el elemento de la cima de la **pila**.	void decapitar ()throws PilaVacia
eliminarPila [4]	Método que recibe una **pila** (que puede tener elementos o no) y la devuelve vacía.	void eliminarPila ()

Tabla 2.1. Especificaciones semánticas y sintácticas del TAD Pila de enteros.

2.2.4. Interfaz del TAD Pila.

El interfaz Pila define los métodos de objeto que utilizaremos con la clase TadPila:

```
import java.io.*;

public interface Pila {
   boolean pilaVacia ();
   void eliminarPila ();
   int cima () throws PilaVacia;
   void apilar (int x);
   int desapilar () throws PilaVacia;
   void decapitar () throws PilaVacia;
   void imprimirPila ();
   void leerPila () throws NumberFormatException, IOException;
   int numElemPila ();
}
```

[2] Se puede producir una situación de excepción cuando el número de elementos de la pila supere determinado valor (lógicamente no podrán ser infinitos). Dicha situación se manifestará al intentar ejecutar la operación de **apilar** sobre una pila en dicho estado.

[3] Se produce una excepción (PilaVacia) si se intenta **desapilar** de una pila vacía. Lo mismo ocurre con las operaciones **cima** y **decapitar**.

[4] Puede resultar innecesario y forzado en Java, pero queremos transmitir la idea de la obligación de crear y destruir los objetos cuando se trabaje en Programación Orientada a Objetos.

2.2.5. Prueba del TAD pila.

2.2.5.1. Condiciones normales.

En el siguiente algoritmo se va a realizar una verificación del funcionamiento elemental (sin situaciones de excepción) del TAD Pila. Primero se inicializa la pila, a continuación se introducen varios elementos en la misma, y por último se sacan dos de los datos, escribiendo los elementos desapilados en la pantalla.

```
public class PruebaPila1 {
   public static void main (String[] args) throws PilaVacia {
      Pila p = new TadPila ();
      int x;

      p.apilar (1);
      p.apilar (2);
      p.apilar (3);
      p.apilar (11);
      p.apilar (15);
      x = p.desapilar ();
      System.out.println ("x = " + x);
      x = p.desapilar ();
      System.out.println ("x = " + x);
      p.eliminarPila ();
   }
}
```

2.2.5.2. Situaciones de excepción.

A continuación se muestra un ejemplo mediante el cual podemos verificar el funcionamiento del TAD frente a situaciones de excepción: introducimos una serie de elementos en la pila, y luego intentamos desapilar uno más de los que tenía la pila originalmente.

```
public class PruebaPila2 {
   public static void main(String[] args) throws PilaVacia {
      Pila pila1 = new TadPila ();
      int i, j;

      System.out.println ();
      for (i = 1; i < 10; i++)
         pila1.apilar (i);
      j = pila1.desapilar ();
      for (i = 1; i < 10; i++)
         j = pila1.desapilar ();
      pila1.eliminarPila ();
   }
}
```

2.2.6. Algoritmos básicos con pilas.

En los siguientes apartados se desarrollan algunos ejemplos elementales que hacen uso del TAD pila siguiendo las especificaciones indicadas en el apartado anterior. En general se tendrán en cuenta las siguientes consideraciones:

- La pila es una estructura muy adecuada para el tratamiento recursivo por lo que se insta a su empleo. Con frecuencia las especificaciones de los enunciados fuerzan a ello tanto por indicación expresa como por la exigencia del cumplimiento simultáneo de las condiciones de realizar un único recorrido (no desapilar/apilar cada elemento más de una vez) y no utilizar estructuras de datos auxiliares.

- Así pues, en general el tratamiento de pilas se hará siguiendo la mecánica *desapilar – llamada recursiva – apilar*. El proceso de los elementos de la pila se realizará, según los casos, a la "ida", a la "vuelta", o parcialmente en ambas fases.

- La condición de terminación "pesimista" supone el cumplimiento (*true*) de la condición *pila.pilaVacia ()*. La terminación anticipada se ejecuta de forma implícita *no realizando llamadas posteriores* cuando se cumpla determinada circunstancia. En este caso se deberá prestar atención especial al tratamiento del último elemento desapilado (si se desea volver a apilarlo o no). El resto de elementos de la pila (desde el actual hasta el fondo) permanece inalterado.

2.2.6.1. Sin terminación anticipada.

2.2.6.1.1. Escribir el contenido de una pila en pantalla.

Se muestra un método (*escribirPila*) que consiste en vaciar la pila (desapilar) durante la fase de "ida". A continuación, se escribe en la pantalla el elemento desapilado y se hace una llamada recursiva. A la vuelta de la recursividad, se deberán apilar los sucesivos valores almacenados localmente en las diferentes instancias de la variable *elem* (local).

El único argumento que necesita el método es la propia *pila*. A continuación se muestra el algoritmo.

```
static void escribirPila (Pila pila) throws PilaVacia {
   int elem;
   if (!pila.pilaVacia ()) {
      elem = pila.desapilar ();
      System.out.println (elem);
      escribirPila (pila);
      pila.apilar (elem);
    }
}
```

2.2.6.1.2. Contar los elementos de una pila.

Se muestra un método que consiste en vaciar la pila (*desapilar*) durante la fase de "ida". En cada llamada se incrementa en una unidad el resultado de la variable local (*resul*) que, en cada paso de la fase de "vuelta", se devuelve como resultado del método. Así mismo se deberán apilar los sucesivos valores almacenados localmente en las diferentes instancias de la variable *elem* (local).

No existe finalización anticipada y la inicialización se produce en el momento de transición del proceso recursivo (al alcanzar la condición de terminación, cuando la pila esté vacía).

El único argumento que necesita el método es la propia *pila*. A continuación se muestra el algoritmo.

```
static int contarPila (Pila pila) throws PilaVacia {
   int elem, resul;
   if (! pila.pilaVacia ()) {
      elem = pila.desapilar ();
      resul = 1 + contarPila(pila);
      pila.apilar (elem);
   }
   else resul = 0;
   return resul;
}
```

2.2.6.1.3. Obtener el duplicado de una pila.

Se trata de obtener un duplicado de la *pila* que se pasa como argumento. En consecuencia el algoritmo necesitará dos argumentos: la pila origen (*pilaO*) y la pila destino (*pilaD*).

En este caso tampoco existe la posibilidad de terminación anticipada sino que seguiremos procesando los datos hasta que se cumpla la condición *pilaO.pilaVacia ()*. La construcción del duplicado de la pila (*pilaD*) se consigue apilando en ella a la vez que se reconstruye la original (en la fase de "vuelta").

A continuación se muestra el algoritmo.

```
static void copiarPila (Pila pilaO, Pila pilaD) throws PilaVacia {
   int elem;
   if (! pilaO.pilaVacia ()) {
      elem = pilaO.desapilar ();
      copiarPila (pilaO, pilaD);
      pilaO.apilar (elem);
      pilaD.apilar (elem);
   }
}
```

Si se deseara obtener un resultado en orden inverso al inicial apilaríamos en la pila destino (*pilaD*) en la fase de "ida", utilizando el siguiente algoritmo:

```
static void copiarPilaI (Pila pilaO, Pila pilaD) throws PilaVacia {
   int elem;
   if (! pilaO.pilaVacia ()) {
      elem = pilaO.desapilar ();
      pilaD.apilar (elem);
      copiarPilaI (pilaO, pilaD);
      pilaO.apilar (elem);
    }
}
```

2.2.6.1.4. Sumergir un elemento en una pila.

Este ejemplo, permite implementar una operación contraria a la naturaleza de la pila. Consiste en un método que introduce un *dato* en el fondo de la pila.

Para poder colocar un elemento en el fondo de la pila, se procede a vaciar la pila, y en el momento que esté vacía se apila el *dato* (transición entre las fases de "ida" y de "vuelta").

```
static void sumergir (Pila pila, int dato) throws PilaVacia {
   int elem;
   if (!pila.pilaVacia ()) {
      elem = pila.desapilar ();
      sumergir (pila, dato);
      pila.apilar (elem);
    }
   else pila.apilar (dato);
}
```

2.2.6.1.5. Invertir los elementos de una pila.

Dada una pila de números enteros, vamos a implementar un método que la devuelva con su contenido invertido. Para ello, utilizaremos el método *sumergir* que hemos visto en el apartado anterior: el método *invertir* consistirá en desapilar un elemento, llamar recursivamente, y a la vuelta de la recursividad, en vez de apilar el elemento en la pila, lo sumergiremos en el fondo de la misma. Nótese que al utilizar dos métodos recursivos, cada uno de los elementos de la pila se desapilará y apilará más de una vez.

```
static void invertir (Pila pila) throws PilaVacia {
   int elem;
   if (!pila.pilaVacia ())   {
      elem = pila.desapilar ();
      invertir (pila);
      sumergir (pila, elem);
   }
}
```

2.2.6.2. Terminación anticipada.

Como ya se ha indicado en algunos casos no será necesario que se alcance la condición de parada en la ejecución del programa (cuando se cumpla determinada circunstancia). Para ello basta con que el algoritmo no ejecute una nueva llamada recursiva, no requiriéndose el empleo de variables de control ni ninguna lógica complementaria.

2.2.6.2.1. Obtener el elemento del fondo de una pila.

Este ejemplo permite implementar una operación contraria a la naturaleza de la pila (ya que debemos llegar a vaciarla para localizar el último elemento).

En el siguiente método se muestra una posible solución que permite identificar, en la fase de "ida", cuál es el elemento del fondo. Cada vez que desapilamos un elemento, verificamos si siguen quedando elementos. En caso afirmativo, hacemos una nueva llamada recursiva y a la "vuelta" apilamos el elemento desapilado. Cuando la pila se ha quedado vacía, acabamos de desapilar el último elemento: guardamos en la variable *dato* el último elemento de la pila (*elem*), y no hacemos más llamadas recursivas. En el caso de que la pila estuviese originalmente vacía devolveríamos un mensaje de error (y como resultado de error devolveríamos -9999).

```
public static int desfondar (Pila p) throws PilaVacia {
   int elem, dato;
   if (!p.pilaVacia ()) {
      elem = p.desapilar ();
      if (!p.pilaVacia ()) {
         dato = desfondar(p);
         p.apilar (elem);
      }
      else dato = elem;
    }
   else {
      System.out.println ("error, la pila está vacia");
      dato = -9999;
    }
   return dato;
}
```

2.2.6.2.2. Comprobar si un elemento pertenece a una pila.

El siguiente ejemplo muestra un método (*esta*) que devuelve un valor booleano dependiendo de que durante el transcurso de exploración de la pila encuentre o no un valor (*dato*) que se pasa como argumento.

En esencia el algoritmo consiste en vaciar la pila (condición pesimista) verificando en cada llamada si el elemento desapilado coincide con el *dato*. En caso negativo se genera una nueva llamada recursiva, pero en caso afirmativo se prepara el valor a devolver por el método (resul = true) y no se realiza una nueva llamada recursiva. Si se llega a vaciar la pila, se entenderá que el elemento buscado no existe y, en consecuencia, el método devolverá el valor *false*.

```
static boolean esta (Pila pila, int dato) throws PilaVacia {
   int elem;
   boolean resul;

   if (!pila.pilaVacia ()) {
      elem = pila.desapilar ();
      if (elem == dato)
         resul = true;
      else resul = esta (pila, dato);
      pila.apilar (elem);
    }
   else resul= false;
   return resul;
}
```

2.2.6.3. Mezclar dos pilas.

La dificultad principal de este tipo de procesos se deriva de la naturaleza "destructiva" de la operación de "lectura" (para conocer el elemento de la cima de una pila hay que "quitarlo" de ella *–desapilar-*). Esto implica que, en un tratamiento recursivo, es posible que algún valor desapilado de alguna de las pilas en instancias anteriores esté durante algún tiempo "pendiente" de ser procesado.

El método, con carácter general, consiste en realizar un tratamiento recursivo cuya estructura es la siguiente:

Fase de ida:

- Se desapilará de una de las dos pilas o de ambas.
- Se realiza el tratamiento específico. Dicho tratamiento deberá incluir el mecanismo de informar a la siguiente instancia, mediante los argumentos apropiados, de qué pila o pilas deberá desapilarse, así como los elementos desapilados en la propia instancia o en alguna instancia anterior (y propagados como argumento en el proceso recursivo),

(*elem1, elem2*)[5]. Para ello se utilizan los argumentos booleanos *apilar1* y *apilar2*. Su interpretación es: un valor *true* indica que el elemento (*elem1|2*) aún no se ha tratado, por lo que queda **pendiente de apilar** en la fase de vuelta (y, en su caso, de tratar) y **no deberá desapilarse** de la pila correspondiente en la siguiente instancia.

- Se realiza una llamada recursiva con los argumentos actualizados durante el tratamiento anterior.
- Se escribe el código que deberá ejecutarse la fase de "vuelta" correspondiente a la instancia actual que, entre otros posibles aspectos, deberá ocuparse de (re)apilar los elementos desapilados en dicha instancia.

Consideración inicial: Los argumentos *apilar1* y *apilar2* deben inicializarse a false (no hay ningún elemento pendiente por tratar). Los valores iniciales de *elem1* y *elem2* no tienen relevancia.

Condición de terminación:

Obsérvese que el hecho de recibir una pila vacía es una condición necesaria para dar por finalizado su tratamiento, pero no suficiente. Es posible que el elemento del fondo de una pila se haya desapilado en instancias anteriores, por lo que la pila se propaga vacía pero aun no se ha tratado su último elemento. Así pues una pila se mantiene "vigente" en el proceso recursivo si:

```
!pila1|2.pilaVacia () || apilar1|2
```

Para facilitar la legibilidad usaremos las variables locales booleanas[6]

```
pend1|2 = (!pila1|2.pilaVacia () || apilar1|2)
```

La cabecera del módulo deberá usar como argumentos:

- *pila1* y *pila2*, de tipo Pila,;
- *apilar1* y *apilar2*, de tipo *boolean*
- *elem1* y *elem2* de tipo *int*

Fase de transición:

Para saber cuándo debemos parar de hacer llamadas recursivas, tenemos dos casos posibles:

- La naturaleza del tratamiento es tal que el proceso finaliza cuando se acaba con el tratamiento de una de las pilas (intersección o modalidad AND). El tratamiento recursivo se mantiene si (*pend1* && *pend2*) == *true* (ambas pilas están pendientes de

[5] En este caso *elem1, elem2* no se utilizan como variables locales tal como se hace en el caso de tratamientos con una sola pila).
[6] Entendidas como "*pendiente*".

tratar). Se deberán apilar (y, en su caso, procesar) los elementos pendientes desapilados en instancias anteriores (identificados mediante la condición *apilar1|2* == true). La fase de "transición" finaliza y se inicia la de "vuelta". (Un posible ejemplo sería: obtener una pila con los elementos **comunes** de otras dos).

- Para que el tratamiento acabe es necesario procesar ambas pilas (unión o modalidad OR). (Por ejemplo, si se desea obtener una pila con los elementos de otras dos). En este caso:
  - Si una pila está procesada (*pend1|2 == false*) y la otra no (*pend2|1 == true*):
    - Se (re)apila, si procede, el elemento desapilado de la pila pendiente de procesar (*apilar2|1*) y se realiza, si procede, el tratamiento oportuno con dicho elemento (*elem2|1*).
    - Se invoca a un (nuevo) tratamiento recursivo que opera únicamente sobre la pila pendiente de procesar.
  - Si ambas pilas están procesadas por completo (*pend1 == false y pend2 == false*), el proceso de "transición" finaliza y se inicia la fase de vuelta.

Fase de vuelta: Desde las correspondientes instancias se realiza el tratamiento adecuado que deberá contemplar, normalmente, la restitución (apilar) de los valores desapilados en la fase de ida.

2.2.6.3.1. Ejemplo1. Realizar la intersección de dos pilas.

Dadas dos pilas (*pila1* y *pila2*) ordenadas ascendentemente desde la cima hacia el fondo (sin elementos repetidos en cada una de las pilas), se desea obtener una nueva pila (*pila3*), también ordenada ascendentemente y sin elementos repetidos, con los elementos comunes de ambas.

Aplicando a este caso el tratamiento genérico, se trata de:

- Fase de "ida". Desapilar si procede (en función del valor de los argumentos *apilar1* y *apilar2*. Existen los siguientes casos:
  - *elem1* > *elem2*. Habrá que (re)apilar *elem2* en *pila2* en la fase de vuelta. La llamada recursiva se hace con *apilar1 = true* y *apilar2 = false*.
  - *elem1* < *elem2*. Tendremos que (re)apilar *elem1* en *pila1* en la fase de vuelta. La llamada recursiva se hace con *apilar1 = false* y *apilar2 = true*.
  - *elem1 == elem2*. En la fase de vuelta se puede apilar cualquiera de los dos elementos (sólo una vez) en *pila3*, y reapilar en *pila1 y pila2*. La llamada recursiva se hace con *apilar1 = false* y *apilar2 = false.*
- Condición de terminación. El proceso finaliza cuando se acaba con el tratamiento de una de las pilas (*pend1* && *pend2*) == *false*.

- Fase de "transición": Si queda algún elemento pendiente en alguna de las pilas (*apilar1 == true* o *apilar2 == true*), habrá que apilarlo en la pila que corresponda.

  Las siguientes figuras muestran un ejemplo.

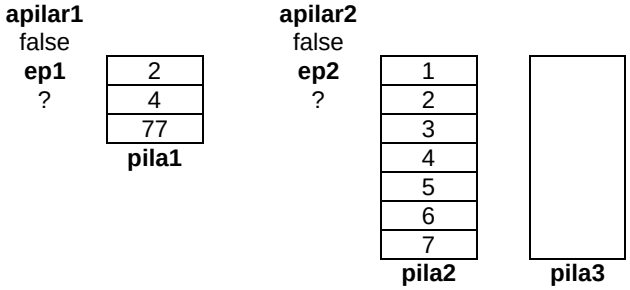

Figura 2.2. Situación de partida.

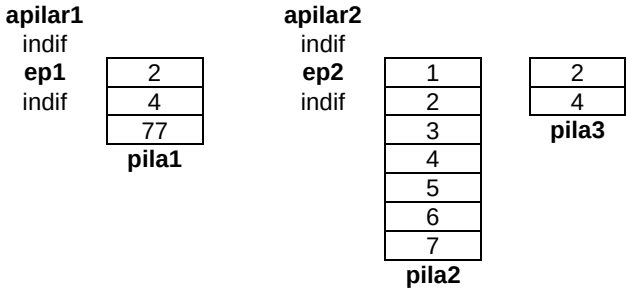

Figura 2.3. Situación final

A continuación aparece el código.

```
static void mezclarPilaAND (Pila pila1, Pila pila2, Pila pila3, boolean apilar1,
boolean apilar2, int elem1, int elem2) throws PilaVacia  {
   boolean pend1, pend2;

   pend1 = !pila1.pilaVacia () || apilar1;
   pend2 = !pila2.pilaVacia () || apilar2;
   if (pend1 && pend2) {
      if (!apilar1)
         elem1 = pila1.desapilar ();
      if (!apilar2)
         elem2 = pila2.desapilar ();
      if (elem1 < elem2) {
         mezclarPilaAND (pila1,pila2,pila3,false,true,elem1,elem2);
         pila1.apilar (elem1);
       }
      else if (elem2 < elem1) {
            mezclarPilaAND (pila1,pila2,pila3,true,false,elem1,elem2);
            pila2.apilar (elem2);
          }
         else {
            mezclarPilaAND (pila1,pila2,pila3,false,false,elem1,elem2);
            pila1.apilar (elem1);
            pila2.apilar (elem2);
            pila3.apilar (elem1);
          }
    }
   else if (apilar1)
            pila1.apilar (elem1);
         else if (apilar2)
            pila2.apilar (elem2);
}
```

2.2.6.3.2. Ejemplo 2. Realizar la unión de dos pilas.

Dadas dos pilas (*pila1* y *pila2*) ordenadas ascendentemente desde la cima hacia el fondo (sin elementos repetidos en cada una de las pilas), se desea obtener una nueva pila (*pila3*), también ordenada ascendentemente y sin elementos repetidos, con los elementos de ambas.

Aplicando a este caso el tratamiento genérico, se trata de:

- Fase de "ida". Desapilar si procede (en función del valor de los argumentos *apilar1* y *apilar2*. Se pueden dar los siguientes casos:
  - *elem1* > *elem2*. Habrá que apilar *elem2* en *pila3* (y reapilar en *pila2*) en la fase de vuelta. La llamada recursiva se hace con *apilar1 = true* y *apilar2 = false.*
  - *elem1* < *elem2*. Tendremos que apilar *elem1* en *pila3* (y reapilar en *pila1*) en la fase de vuelta. La llamada recursiva se hace con *apilar1 = false* y *apilar2 = true.*
  - *elem1 == elem2*. En la fase de vuelta se puede apilar cualquiera de los dos elementos (solo una vez) en *pila3*, y reapilar en *pila1 y pila2*. La llamada recursiva se hace con *apilar1 = false* y *apilar2 = false*.
- Condición de terminación. Para que el tratamiento acabe es necesario procesar ambas pilas (*pend1* || *pend2*) == *false*
- Fase de "transición"
  - Si se ha "terminado" con *pila1* (*pend1 == true*) y no con *pila2* (*pend2 == false*).
    - Se aplical método *copiarPila* con efectos sobre el estado actual de *pila2.*
    - Se restaura, si procede, el último valor desapilado de *pila1*.
    - Se inicia la fase de "vuelta"
  - Si se ha "terminado" con *pila2* (*pend2 == true*) y no con *pila1* (*pend1= = false*).
    - Se aplica el método *copiarPila* con efectos sobre el estado actual de *pila1*.
    - Se restaura, si procede, el último valor desapilado de *pila2*.
    - Se inicia la fase de "vuelta".
  - Si se ha "terminado" con ambas pilas (*pend1 == false* y *pend2 == false*).
    - Se inicia la fase de vuelta.

  Las siguientes figuras muestran un ejemplo:

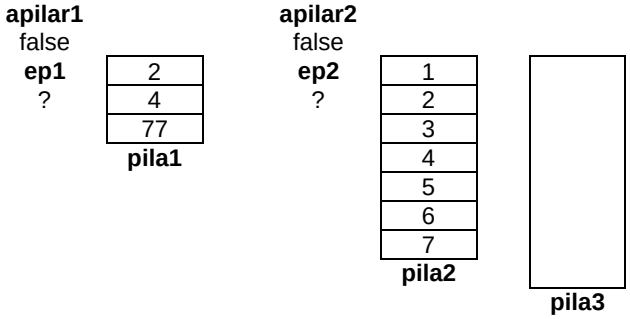

Figura 2.4. Situación de partida.

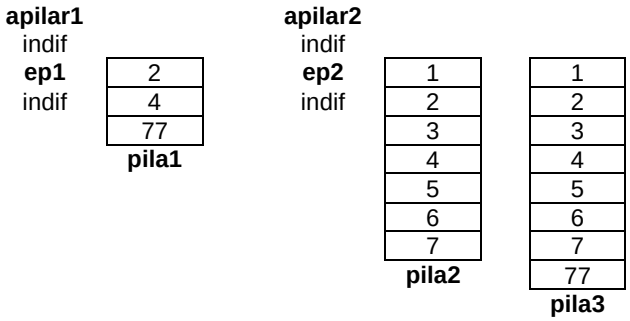

Figura 2.5. Situación final

A continuación se muestra el código.

```
static void copiarPila (Pila pilaO, Pila pilaD) throws PilaVacia  {
   int elem;
   if (! pilaO.pilaVacia ()) {
      elem = pilaO.desapilar ();
      copiarPila (pilaO, pilaD);
      pilaO.apilar (elem);
      pilaD.apilar (elem);
    }
}
static void mezclarPilaOR (Pila pila1, Pila pila2, Pila pila3, boolean apilar1,
boolean apilar2, int elem1, int elem2) throws PilaVacia {
   boolean pend1, pend2;
   pend1 = !pila1.pilaVacia () || apilar1;
   pend2 = !pila2.pilaVacia () || apilar2;
   if (pend1 && pend2) {
      if (!apilar1)
         elem1 = pila1.desapilar ();
      if (!apilar2)
         elem2 = pila2.desapilar ();
      if (elem1 < elem2) {
         mezclarPilaOR (pila1, pila2, pila3, false, true, elem1, elem2);
         pila1.apilar (elem1);
         pila3.apilar (elem1);
       }
      else if (elem2 < elem1) {
               mezclarPilaOR (pila1,pila2,pila3,true,false,elem1,elem2);
               pila2.apilar (elem2);
               pila3.apilar (elem2);
             }
           else {
               mezclarPilaOR (pila1,pila2,pila3,false,false,elem1,elem2);
               pila1.apilar (elem1);
               pila2.apilar (elem2);
               pila3.apilar (elem1);
             }
    }
   else if (pend1 && !pend2) {
               copiarPila (pila1, pila3);
               if (apilar1) {
                  pila1.apilar (elem1);
                  pila3.apilar (elem1);
                }
             }
           else if (pend2 && !pend1) {
                    copiarPila (pila2, pila3);
                    if (apilar2) {
                       pila2.apilar (elem2);
                       pila3.apilar (elem2);
                     }
                  }
}
```

2.2.6.4. Mezcla de pilas con terminación anticipada.

Dadas dos pilas (*pila1* y *pila2*) ordenadas ascendentemente desde la cima hacia el fondo (sin elementos repetidos en cada una de las pilas), se desea realizar un método booleano que indique si los elementos de *pila2* están contenidos en *pila1* o no.

Esta situación se puede resolver a partir de la consideración de mezcla de pilas en la modalidad AND (termina cuando se ha procesado por completo cualquiera de las pilas).

En este caso se produce una situación de terminación anticipada cuando, en su caso, aparece por primera vez un elemento de *pila2* que no está en *pila1* (*elem1 > elem2*). El proceso debe terminar y el método devuelve false.

En caso contrario (terminación "pesimista") pueden darse las siguientes circunstancias:

- Se ha procesado completa *pila1* (*pend1 == false*) pero *pila2* no (*pend2 == true*). Aún quedan elementos en *pila2* por tanto no cumple la condición de "estar contenida" en *pila1*, y *en consecuencia* el resultado será *false*.
- Se ha procesado *pila2* completa (*pend2 == false*) y pero no *pila1* (*pend1 == true*). Por tanto, *pila2* estará contenida en *pila1; y en consecuencia* el resultado será *true*.
- Se ha conseguido llegar a la terminación "pesimista" (se han procesado completamente ambas pilas: *pend1 == false* y *pend2 == false*). Por lo tanto, *pila2* está contenida en *pila1* y el método devolverá el valor *true*.

  Aplicando a este caso el tratamiento genérico, se trata de:

- Fase de "ida". Desapilar si procede (en función del valor de los argumentos *apilar1* y *apilar2*). Se pueden dar los siguientes casos:
  - *elem1 > elem2*. Hay que reapilar *elem2* en *pila2* en la fase de vuelta. Es imposible que la *pila2* se encuentre contenida en la *pila1*, por lo que habrá que devolver *false* como resultado.
  - *elem1 < elem2*. Hay que reapilar *elem1* en *pila1* en la fase de vuelta. La llamada recursiva se hace con *apilar1 = false* y *apilar2 = true*.
  - *elem1 == elem2*. En la fase de vuelta se reapilan los dos elementos en *pila1 y pila2*. La llamada recursiva se hace con *apilar1 = false* y *apilar2 = false*.
- Condición de terminación. El proceso finaliza cuando se acaba con el tratamiento de una de las pilas (*pend1* && *pend2*) == *false*.
- Fase de "transición"
  - Si queda algún elemento pendiente de tratar en alguna de las pilas (*apilar1 == true* o *apilar2 == true*), habrá que apilarlo en la pila que corresponda y asignar el resultado correspondiente al método.

El código se muestra a continuación:

```
 static boolean contenida (Pila pila1, Pila pila2, boolean apilar1, boolean
apilar2, int elem1, int elem2) throws PilaVacia {
   boolean pend1, pend2, resul;

   pend1 = !pila1.pilaVacia () || apilar1;
   pend2 = !pila2.pilaVacia () || apilar2;
   if (pend1 && pend2) {
      if (!apilar1)
         elem1 = pila1.desapilar ();
      if (!apilar2)
         elem2 = pila2.desapilar ();
      if (elem1 < elem2) {
         resul = contenida (pila1, pila2, false, true, elem1, elem2);
         pila1.apilar (elem1);
       }
      else if (elem2 < elem1) {
               resul = false;
               pila2.apilar (elem2);
               pila1.apilar (elem1);
            }
           else{
                  resul = contenida (pila1,pila2,false,false,elem1,elem2);
                  pila1.apilar (elem1);
                  pila2.apilar (elem2);
            }
    }
   else if (apilar1) {
           resul = true;
           pila1.apilar (elem1);
         }
        else if (apilar2) {
                  resul = false;
                  pila2.apilar (elem2);
                }
               else resul = true;
   return resul;
}
```

~~2.3.~~

~~2.4.~~2.3. TAD COLA DE NÚMEROS ENTEROS.

~~2.4.1.~~2.3.1. Concepto.

Una cola es una agrupación de elementos de determinada naturaleza o tipo (datos de personas, números, procesos informáticos, automóviles, etc.) entre los que existe definida una relación de orden (**estructura de datos**). En función del tiempo, algunos elementos pueden llegar a la cola o salir de ella (**operaciones / acciones**). En consecuencia el estado de la cola varía.

Una cola presenta comportamiento FIFO (*First Input First Output*) y se respeta como criterio de ordenación el momento de la llegada: el primer elemento de la cola, será el que primero llegó a ella y, en consecuencia, el primero que saldrá, y así sucesivamente.

~~2.4.2.~~2.3.2. Modelo gráfico.

Podríamos representar gráficamente una cola según aparece en la Figura 2.6: una estructura de datos horizontal, en la que los elementos se insertan por el extremo derecho, y se extraen por la parte izquierda.

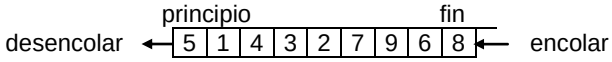

Figura 2.6. Modelo gráfico de Cola

~~2.4.3.~~2.3.3. Especificaciones

En la Tabla 2.2., que aparece en la página siguiente, se definen las especificaciones semánticas y sintácticas del TAD cola de enteros. Utiliza el tipo de datos TadCola (cola de números enteros). Cuando se desee utilizar el tad cola, habrá que poner la siguiente sentencia en la primera línea del programa, después de la cabecera (o bien utilizarlo dentro del mismo paquete –*package tadCola*–):

```
import tadCola.*;
```

Operación	Especificación semántica	Especificación sintáctica
encolar [7]	Método que entrega un elemento (x) para que quede incorporado al final de la **cola**.	void encolar (int x)
desencolar [8]	Método que elimina el elemento de la **cola** que ocupa el primer lugar, devolviéndolo como resultado.	int desencolar () throws ColaVacia
colaVacia	Método que al ejecutarse devuelve *true* si la **cola** está vacía, y *false* en caso contrario.	boolean colaVacia ()
leerCola	Método mediante el que se produce la carga inicial de elementos de la **cola**.	void leerCola () throws NumberFormatException, IOException
imprimirCola	Método que muestra en el dispositivo de salida (pantalla) el contenido actual de la **cola**.	void imprimirCola ()
invertirCola	Método que devuelve la **cola** con sus elementos invertidos	void invertirCola () throws ColaVacia
numElemCola	Método que devuelve el número de elementos de la **cola**.	int numElemCola ()
primero	Método que devuelve el primer elemento de la **cola** sin desencolarlo.	int primero () throws ColaVacia
quitarPrimero	Método que elimina el primer elemento de la **cola**.	void quitarPrimero () throws ColaVacia
eliminarCola	Método que recibe una **cola** (que puede tener elementos o no) y la devuelve vacía.	void eliminarCola ()

Tabla 2.2. Especificaciones semánticas y sintácticas del TAD Cola de enteros.

~~2.4.4.~~2.3.4. Interfaz del TAD Cola.

```
import java.io.IOException;
public interface Cola {
   boolean colaVacia ();
   void eliminarCola ();
   int primero () throws ColaVacia;
   void encolar (int x);
   int desencolar () throws ColaVacia;
   void quitarPrimero () throws ColaVacia;
   void mostrarEstadoCola ();
   void imprimirCola ();
   void leerCola () throws NumberFormatException, IOException;
   int numElemCola ();
   void invertirCola () throws ColaVacia;
}
```

[7] Se puede producir una situación de excepción cuando el número de elementos de la cola supere determinado valor (lógicamente no podrán ser infinitos) y se intente **encolar** sobre una cola en dicho estado.

[8] Se produce una excepción (ColaVacia) si se intenta **desencolar** de una cola vacía. Lo mismo ocurre con las operaciones **primero** y **quitarPrimero**.

~~2.4.5.~~2.3.5. Prueba del TAD Cola.

~~2.4.5.1.~~2.3.5.1. Condiciones normales.

El siguiente algoritmo simplemente se trata de introducir y extraer elementos de la cola sin provocar situaciones de excepción.

```
public class PruebaCola1 {
   public static void main (String [ ] args) throws ColaVacia {
      Cola cola1 = new TadCola ();
      int Elem;
      cola1.encolar (8);
      cola1.encolar (7);
      cola1.encolar (9);
      cola1.encolar (11);
      Elem = cola1.desencolar ();
      Elem = cola1.desencolar ();
      System.out.println ("Acaba de salir el numero "+Elem);
      cola1.eliminarCola ();
   }
}
```

~~2.4.5.2.~~2.3.5.2. Situaciones de excepción.

En el siguiente ejemplo se verifica el funcionamiento del TAD frente a situaciones de excepción.

```
public class PruebaCola2 {
   public static void main (String [ ] args) throws ColaVacia {
      Cola cola1 = new TadCola ();
      int i, j;
      for (i = 1; i < 10; i++)
         cola1.encolar (i);
      j = cola1.desencolar ();
      System.out.println ("Hemos sacado " + j);
      for (i = 1; i < 10; i++) {
         j = cola1.desencolar ();
         System.out.println("Hemos sacado " + j);
      }
      cola1.eliminarCola ();
   }
}
```

~~2.4.6.~~2.3.6. Algoritmos básicos con colas.

La filosofía general para la solución para el tratamiento de pilas, mediante técnicas recursivas, basada en un esquema de *desapilar – llamada recursiva – apilar* con una condición de finalización vinculada a la situación de estructura vacía no dará en general buenos resultados con colas, dado que devolvería la estructura con sus elementos cambiados de orden respecto a la situación inicial (devolverá la cola invertida).

Tampoco sería válido un planteamiento del tipo de *desencolar – encolar – llamada recursiva*, pues nunca se alcanzaría la condición de finalización.

En consecuencia para basarse en la primera idea resultará, casi siempre, necesario ejecutar un proceso complementario de invertir los elementos de la cola (como alternativa podría usarse una estructura de datos auxiliar).

El proceso se simplifica notablemente si se conoce a priori el número inicial de elementos de la cola pues a partir de este valor se podría utilizar la segunda idea **utilizando como condición de finalización el valor de una variable**, inicialmente con el valor del **número de elementos** de la cola, que se decrementaría con cada llamada al algoritmo. También podría procederse en este caso a un tratamiento iterativo.

~~2.4.6.1.~~2.3.6.1. Invertir el orden de los elementos de una cola.

Si deseamos invertir una cola, y suponiendo que no se conoce el número de elementos que contiene, ni tenemos invertirCola entre las operaciones del TAD, se propone una solución recursiva, acorde con la tendencia a invertirla que produce este tipo de tratamiento.

```
static void invertir (Cola cola) throws ColaVacia {
   int elem;

   if (!cola.colaVacia ()) {
      elem = cola.desencolar ();
      invertir (cola);
      cola.encolar (elem);
   }
}
```

~~2.4.6.2.~~2.3.6.2. Contar los elementos de una cola.

Si hay que hacer un método que cuente los elementos de una cola, evidentemente no se conoce a priori el número de elementos. La única posibilidad para realizar el método es un tratamiento recursivo que, aunque cuente correctamente el número de elementos de la cola, los devolvería en el orden contrario al inicial. Se necesita, pues, una ejecución (antes o después) del método visto en el apartado anterior (invertir) fuera del tratamiento recursivo.

```
static void invertir (Cola cola) throws ColaVacia {
   int elem;

   if (!cola.colaVacia ()) {
      elem = cola.desencolar ();
      invertir (cola);
      cola.encolar (elem);
    }
}
static int contarCola (Cola cola) throws ColaVacia {
   int elem, resul;

   if (! cola.colaVacia ()) {
      elem = cola.desencolar ();
      resul = 1 + contarCola (cola);
      cola.encolar (elem);
    }
   else resul = 0;
   return resul;
}
static int cuentaCola (Cola cola) throws ColaVacia {
   invertir (cola);
   return contarCola (cola);
}
```

~~2.4.6.3.~~2.3.6.3. Obtención una cola a partir de otra.

Siempre que vayamos a trabajar con colas, tenemos tres formas posibles de hacerlo:

- Si no conocemos el número de elementos de la cola, necesariamente tendremos que realizar un tratamiento recursivo siguiendo el esquema *desencolar–tratamiento recursivo–encolar*. El problema que tenemos es el que ya se ha planteado previamente: este tratamiento deja la cola invertida. Por lo tanto, posteriormente habrá que invertir la cola resultante.

- Cuando conocemos el número de elementos de la cola, podemos optar entre las dos siguientes posibilidades:

  - Hacer un tratamiento recursivo, utilizando el número de elementos de la cola para fijar la condición de parada de la recursividad. Tendremos que realizar el tratamiento *desencolar–encolar* antes de la recursividad.

  - Realizar un tratamiento iterativo, por medio de un bucle *for*.

  Utilizando cualquiera de estos dos métodos, los elementos de la cola se quedarán en el mismo orden inicial.

Como ejemplo, en los siguientes subapartados se realiza la copia de una cola sobre otra siguiendo los tres métodos propuestos.

~~2.4.6.3.1.~~2.3.6.3.1. Sin conocer el número de elementos.

Necesariamente deberá realizarse mediante un tratamiento recursivo que devolverá la cola original (*ColaO*) con sus elementos en orden contrario respecto al inicial. Para restaurarlo se ejecutará una llamada (no recursiva) al método invertir.

```
static void copiar (Cola colaO, Cola colaD) throws ColaVacia {
   int elem;
   if (!colaO.colaVacia ()) {
      elem = colaO.desencolar();
      colaD.encolar (elem);
      copiar (colaO, colaD);
      colaO.encolar (elem);
    }
}
static void copiaRecursiva1 (Cola colaO, Cola colaD) throws ColaVacia {
   copiar (colaO,colaD);
   invertir (colaO);
}
```

Si lo que se desea es obtener una cola (*ColaD*) ordenada de forma inversa a la original (*ColaO*) bastará con encolar en la fase de "vuelta".

```
static void copiarInvertido (Cola colaO, Cola colaD) throws ColaVacia {
   int elem;
   if (!colaO.colaVacia ()) {
      elem = colaO.desencolar();
      copiarInvertido (colaO, colaD);
      colaD.encolar (elem);
      colaO.encolar (elem);
    }
}
static void copiaRecursivaInvertida (Cola colaO, Cola colaD) throws ColaVacia {
   copiarInvertido (colaO, colaD);
   colaO.invertirCola ();
}
```

~~2.4.6.3.2.~~2.3.6.3.2. Conociendo el número de elementos. Técnica recursiva.

Como ya se ha indicado, es necesario utilizar como condición de parada el valor del argumento *n* (número de elementos de la cola) que se decrementa en una unidad en cada llamada.

```
static void copiaRecursiva2 (Cola colaO, Cola colaD, int n) throws ColaVacia {
   int elem;
   if (n > 0) {
      elem = colaO.desencolar();
      colaD.encolar (elem);
      colaO.encolar (elem);
      copiaRecursiva2 (colaO, colaD, n-1);
   }
}
```

Análogamente al caso anterior, para obtener un duplicado con los elementos invertidos se utilizaría la variante siguiente.

```
static void copiaRecursiva2Invertida (Cola colaO, Cola colaD, int n) throws
ColaVacia {
   int elem;

   if (n > 0) {
      elem = colaO.desencolar ();
      colaO.encolar (elem);
      copiaRecursiva2Invertida (colaO, colaD, n-1);
      colaD.encolar (elem);
   }
}
```

~~2.4.6.3.3.~~2.3.6.3.3. Conociendo el número de elementos. Técnica iterativa.

De forma similar se utiliza la variable *N* (número de elementos de la cola) como condición de fin de un bucle *for*.

```
static void copiaIterativa (Cola colaO, Cola colaD) throws ColaVacia {
   int elem, i, n;

   n = colaO.numElemCola ();
   for (i = 1; i <= n; i++) {
      elem = colaO.desencolar ();
      colaD.encolar (elem);
      colaO.encolar (elem);
   }
}
```

Si lo que se pretende es obtener la cola resultante (*colaD*) con sus elementos en orden contrario al inicial (*colaO*) sería necesario invertir posteriormente el resultado (ejecución del método i*nvertir*) una vez ejecutado el tratamiento iterativo.

```
static void copiaIterativaInvertida (Cola colaO, Cola colaD) throws ColaVacia {
   int elem, i, n;

   n = colaO.numElemCola ();
   for (i = 1; i <= n; i++) {
      elem = colaO.desencolar ();
      colaD.encolar (elem);
      colaO.encolar (elem);
    }
   colaD.invertirCola ();
}
```

~~2.4.6.4.~~2.3.6.4. Insertar un elemento al principio de una cola.

Se trata de introducir en la *cola* un *dato,* pasado como argumento, que no se ubicará al final de la misma (tal como resultaría como consecuencia de ejecutar la operación *encolar (cola, dato)*), sino al principio de la misma. La Figura 2.8 muestra como ejemplo el resultado de ejecutar el algoritmo introduciendo el *dato* 77 en la cola cuya situación inicial es la representada en la Figura 2.7.

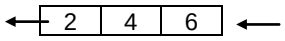

Figura 2.7. Situación inicial de la cola.

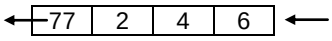

Figura 2.8. Resultado después de introducir el 77.

Se resolverá de manera iterativa (conociendo el número de elementos). Para ello, la inserción inicial del dato al final de la cola se produce con anterioridad a la ejecución de la iteración (pero una vez contado el número de elementos que tiene la cola original). Se propone al alumno la realización del mismo ejemplo siguiendo los otros dos métodos propuestos.

```
static void insertarPrincipioIterativo (Cola cola, int dato) throws ColaVacia {
   int elem, i, n;
   n = cola.numElemCola ();
   cola.encolar (dato);
   for (i = 1; i <= n; i++) {
      elem = cola.desencolar();
      cola.encolar (elem);
   }
}
```

~~2.4.6.5.~~2.3.6.5. Terminación anticipada.

Como ya se ha indicado, en algunos casos no será necesario que en la ejecución del programa se procese toda la estructura produciendo, en consecuencia, una terminación anticipada. La solución depende de la técnica empleada: iterativa o recursiva si se conoce el número de elementos o, necesariamente recursiva si no se conoce.

En cualquier caso deberá asegurarse que la estructura se devuelve en el orden correcto, lo que implica que todos los elementos deberán ser desencolados y encolados (además, cuando no se conoce el número de elementos, deberán invertirse tanto la cola completa como la parte de ella que no se haya tratado en el momento de la terminación anticipada). Pese a ello la terminación anticipada tiene interés en general, dado que se evitan copias innecesarias en la memoria en la fase de "ida" y en particular cuando dicho tratamiento resulte complejo.

A continuación se muestran ejemplos de un método (*esta*) que devuelve un valor booleano dependiendo de que durante el transcurso de exploración de la cola encuentre o no un valor (*dato*) que se pasa como argumento. Por último, se ve un método que permite obtener el elemento final de una cola.

~~2.4.6.5.1.~~2.3.6.5.1. Verificar si hay un elemento sin conocer el número de elementos.

En esencia el algoritmo consiste en vaciar la cola (condición pesimista) verificando en cada llamada si el elemento desencolado coincide con el *dato*. En caso negativo se genera una nueva llamada recursiva pero en caso afirmativo se prepara el valor a devolver por el método (*resul*=true) y no se realiza una nueva llamada recursiva. Deberá asegurarse que se devuelve a la cola el último elemento desencolado. Si se alcanza la condición de terminación se entenderá que el elemento buscado no existe y, en consecuencia, el método devolverá el valor false.

Un problema adicional es que el tratamiento recursivo produce el desorden parcial de los elementos de la cola. Es decir, en caso de que se produzca terminación anticipada la cola tendería a quedar con el siguiente aspecto:

- El bloque de elementos previo al dato encontrado quedaría al final de la cola y con sus elementos en orden inverso.
- El bloque de elementos posterior al dato encontrado quedaría al principio de la cola y con sus elementos en el orden inicial.
- El dato buscado estaría situado entre ambos bloques.

Por ejemplo, si en la cola indicada en la Figura 2.9 se busca el elemento de valor 12, el método devolvería correctamente el valor true pero dejaría sus elementos en la situación de la Figura 2.10.

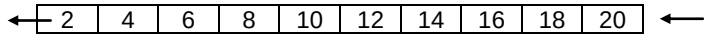

Figura 2.9. Situación inicial de la cola. Se busca el valor 12.

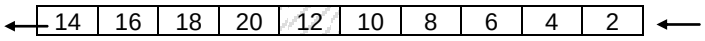

Figura 2.10. Valor 12 encontrado. Situación actual de los elementos de la cola.

Por lo tanto, para evitar este problema, se deberá utilizar un método externo (previo o posterior) de inversión de la cola inicial así como otra inversión de lo que queda de la cola una vez encontrado el dato (antes de volver a encolarlo). A continuación se muestra el código.

```
static boolean estaAux (Cola cola, int dato) throws ColaVacia {
   int elem;
   boolean resul;
   if (! cola.colaVacia ()) {
      elem = cola.desencolar ();
      if (elem == dato) {
         resul = true;
         cola.invertirCola ();
       }
      else resul = estaAux (cola, dato);
      cola.encolar (elem);
    }
   else resul = false;
   return resul;
}
static boolean esta (Cola cola, int dato) throws ColaVacia {
   cola.invertirCola ();
   return estaAux (cola, dato);
}
```

~~2.4.6.5.2.~~2.3.6.5.2. Conociendo el número de elementos. Técnica recursiva.

El algoritmo es más sencillo que en el caso anterior. El proceso es un conjunto *desencolar – encolar* (controlado por el argumento que indica inicialmente el valor de los elementos de la cola) en el que se pregunta si el elemento desencolado coincide con el *dato* que se busca. En caso negativo se realiza una nueva llamada recursiva y en caso afirmativo se asigna el valor que devolverá el método: *true* (no se realizan posteriores llamadas recursivas pero habrá que "recolocar" completamente la cola, por medio del método auxiliar *recorrer*). El cumplimiento de la condición pesimista (el módulo se ha ejecutado tantas veces como elementos tiene la cola) implica que el método devuelve el valor false.

A continuación se muestra una posible solución.

```
static boolean recorrer (Cola cola, int n) throws ColaVacia {
   boolean resul;
   int elem;
   if (n > 0) {
      elem = cola.desencolar ();
      cola.encolar (elem);
      resul = recorrer (cola, n-1);
    }
   else resul = true;
   return resul;
}
static boolean estaRecursivo (Cola cola, int n, int dato) throws ColaVacia {
   int elem;
   boolean resul;
   if (n > 0) {
      elem = cola.desencolar ();
      cola.encolar (elem);
      if (elem == dato)
         resul = recorrer (cola, n-1);
         else resul = estaRecursivo (cola, n-1, dato);
    }
   else resul = false;
   return resul;
}
```

~~2.4.6.5.3.~~2.3.6.5.3. Conociendo el número de elementos. Técnica iterativa.

Si tenemos el número de elementos, y queremos realizar el tratamiento de manera iterativa, el algoritmo será similar al caso anterior, si bien aquí sí es necesario complicar la lógica para facilitar la terminación anticipada *(n > 0) && (!resul*). Una vez que se ha encontrado el dato, hay que continuar desencolando y encolando hasta terminar de dar la vuelta completa a la cola (bucle *for (i = 1; i<= n; i++)*), como se puede ver en la siguiente solución:

```
static boolean estaIterativo (Cola cola, int dato) throws ColaVacia {
   int n, elem, i;
   boolean resul;

   resul = false;
   n = cola.numElemCola ();
   while ((n > 0) && (!resul)) {
      elem = cola.desencolar ();
      cola.encolar (elem);
      if (elem == dato)
         resul = true;
      n = n-1;
    }
   for (i = 1; i <= n; i++) {
      elem = cola.desencolar ();
      cola.encolar (elem);
    }
   return resul;
}
```

~~2.4.6.5.4.~~2.3.6.5.4. Obtener el elemento del final de una cola.

Lo que se propone en este apartado es conseguir un efecto similar al de la operación desencolar pero actuando sobre el extremo opuesto, es decir, no queremos sacar el primer elemento de la cola, sino el último que hemos insertado. Se muestra como ejemplo de manera recursiva, sin conocer el número de elementos. Se propone al alumno la realización del mismo ejemplo siguiendo los otros dos métodos propuestos.

Es un caso similar al explicado en el apartado de pilas (*desfondar*) con la salvedad de que aquí es necesario restaurar el orden inicial en los elementos de la cola. Para ello se utiliza la operación *InvertirCola* una vez ejecutado el tratamiento recursivo. Al contrario de lo que ocurría en el apartado 2.3.5.6.1., en este caso excepcionalmente no hay que invertir lo que queda de cola en la transición, dado que en este momento la cola está vacía.

```
static int quitarUltimo (Cola cola) throws ColaVacia {
   int elem, dato = -9999;

   if (!cola.colaVacia ()) {
      elem = cola.desencolar ();
      if (! cola.colaVacia ()) {
         dato = quitarUltimo (cola);
         cola.encolar (elem);
       }
      else dato = elem;
    }
   return dato;
}
static int quitarUltimoRecursivo1 (Cola cola) throws ColaVacia {
   int resul;

   resul = quitarUltimo (cola);
   System.out.println ("resultado "+resul);
   cola.invertirCola ();
   return resul;
}
```

~~2.4.6.6.~~2.3.6.6. Mezclar dos colas.

De forma análoga a lo explicado en el caso de mezcla de pilas, el problema específico de este tipo de procesos es la naturaleza "destructiva" de la operación desencolar.

El problema puede resolverse mediante los tres enfoques habituales con colas:

- No se conoce el número de elementos. Se deberá utilizar necesariamente un tratamiento recursivo similar al explicado con pilas, pero teniendo en cuenta la circunstancia de que las colas pueden quedar total o parcialmente desordenadas y por tanto habrá que utilizar el módulo *invertir*. Se deja su resolución como ejercicio propuesto al alumno.
- Se conoce el número de elementos (*n1* y *n2*). Se puede hacer:
  - De manera iterativa. Se deja como ejercicio propuesto al alumno
  - De manera recursiva. Se explica a continuación.

Explicación de la técnica.

La técnica es ligeramente diferente a la explicada en el caso de la mezcla de pilas. En este caso hay que desencolar "por adelantado". Es decir: en la instancia actual se desencola de una o de ambas colas y se pasan como argumentos los elementos desencolados (*elem1*|*elem2*). En caso de desencolar de una sola cola, el elemento de la otra se propaga a la instancia siguiente.

Este tipo de tratamiento supone una singularidad en la primera llamada: se necesita un módulo (de "lanzamiento"), no recursivo, que desencola y encola el primer elemento de cada una de las colas y los pasa como argumentos al módulo recursivo.

Condición de terminación.

En este caso el control de la condición de finalización se realiza mediante los argumentos *n1* y *n2* que representan inicialmente el número de elementos de las respectivas colas. Dichos argumentos deberán pasarse por valor y decrementarse en cada llamada recursiva. La condición ((*n1 == 0)* || (*n2 == 0))* significa que se ha desencolado el último elemento de la cola correspondiente (condición necesaria) **pero no suficiente, dado que no se asegura que dichos elementos se hayan tratado** (lo que se deberá hacer en la fase de"transición"). Así pues el proceso recursivo tiene lugar si (*n1 >= 0 && n2 >= 0*)[9]

Fase de "ida":

- Se realiza el tratamiento específico con los elementos (*elem1* y *elem2*) que se reciben como argumentos.
- Se desencola-encola de una de las colas o de ambas.
- Se realiza una llamada recursiva.

Fase de "transición":

- Se realiza, si procede, el tratamiento específico del ejercicio en cuestión.
- Se restaura, si procede, el orden establecido de las colas.

Fase de "vuelta":

- Se realiza, si procede, el tratamiento específico del ejercicio en cuestión.
- En general no será necesario realizar la operación de encolar dado que ya se ha hecho en la fase de "ida" (mecanismo desencolar-encolar).

~~2.4.6.6.1.~~2.3.6.6.1. Ejemplo 1. Realizar la intersección de dos colas.

En este ejemplo se parte de dos colas (*cola1* y *cola2*) ordenadas ascendentemente desde el punto de salida hacia el de llegada para obtener una nueva cola (*cola3*) con los elementos comunes de ambas ordenados en el mismo sentido. La Figura 2.11. muestra un ejemplo.

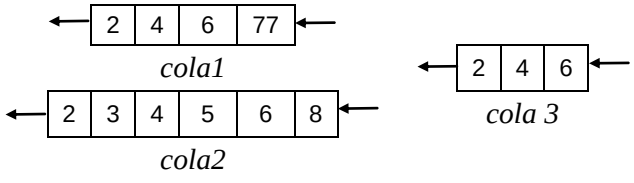

Figura 2.11.Ejemplo de obtención de una cola con los elementos comunes de otras dos

[9] O, dicho en otros términos: finaliza si: (*n1 < 0 || n2 < 0*)

Para resolver este ejemplo, utilizaremos la técnica recursiva conociendo el número de elementos. Como en este caso queremos introducir en la *cola3* todos los elementos comunes de la *cola1* y la *cola2*, la condición de parada de la recursividad será cuando hayamos tratado todos los elementos de una de las dos colas (*n1 < 0 || n2 < 0*).

El diseño propuesto requiere el empleo de instrucciones externas al algoritmo recursivo: Previamente es necesario desencolar-encolar de ambas colas para disponer de los valores iniciales de *elem1* y *elem2*.

Así pues la cabecera del método recursivo quedaría:

```
static void mezclaAndR2 (Cola cola1, Cola cola2, Cola cola3, int elem1, int elem2,
int n1 , int n2)
```

Fase de "ida":

- Se comparan los elementos procedentes de ambas colas. Tanto *elem1* como *elem2* proceden de operaciones de desencolar-encolar ejecutadas en instancias anteriores por lo que se procederá a encolar el menor de ellos (o uno cualquiera si son iguales) en *cola3* (como *cola3* debe tener sus elementos ordenados en el mismo sentido que *cola1* y *cola2*, los elementos se encolarán en *cola3* antes de realizar la correspondiente llamada recursiva).
- Se desencola-encola de las cola/s pertinentes si es posible (*n1 |2 >* 0), pasando así *elem1|2* a la instancia siguiente.

Fase de "transición":

- Si una de las colas tiene elementos por tratar y la otra no, se llama al método auxiliar *reOrdenar*, que recoloca la cola que tiene elementos pendientes por tratar.

Fase de "vuelta":

- No hay que realizar ningún tratamiento específico, ni es necesario realizar la operación de encolar dado que ya se ha hecho en la fase de "ida" (mecanismo desencolar-encolar).

A continuación se muestra el algoritmo que resuelve el problema

```
static void reOrdenar (Cola cola, int n) throws ColaVacia {
  int elem;

  if (n > 0) {
    elem = cola.desencolar ();
    cola.encolar (elem);
    reOrdenar (cola, n-1);
    }
}
static void mezclaAndR2 (Cola cola1, Cola cola2, Cola cola3, int elem1, int elem2,
int n1 , int n2) throws ColaVacia {

  if ((n1 >= 0) && (n2 >= 0))
    if (elem1 < elem2) {
       if (n1 > 0) {
         elem1 = cola1.desencolar ();
         cola1.encolar (elem1);
        }
       mezclaAndR2 (cola1,cola2,cola3,elem1,elem2,n1-1,n2);
     }
    else if (elem1 > elem2) {
       if (n2 > 0) {
         elem2 = cola2.desencolar ();
       cola2.encolar (elem2);
       }
       mezclaAndR2 (cola1,cola2,cola3,elem1,elem2,n1,n2-1);
      }
      else {
         cola3.encolar (elem1);
         if (n1 > 0) {
            elem1 = cola1.desencolar ();
            cola1.encolar (elem1);
          }
         if (n2 > 0) {
            elem2 = cola2.desencolar();
            cola2.encolar (elem2);
          }
         mezclaAndR2 (cola1,cola2,cola3,elem1,elem2,n1-1,n2-1);
       }
  else {
    if (n1 >= 0)
       reOrdenar (cola1, n1 );
    else if (n2 >= 0)
            reOrdenar (cola2, n2);
    }
}

static void mezclarColasAndR2 (Cola cola1,Cola cola2, Cola cola3) throws ColaVacia {
  int elem1, elem2, n1 , n2;

  n1 = cola1.numElemCola ();
  n2 = cola2.numElemCola ();
  if ((n1 > 0) && (n2 > 0)) {
    elem1 = cola1.desencolar ();
    cola1.encolar (elem1);
    elem2 = cola2.desencolar ();
    cola2.encolar (elem2);
    mezclaAndR2(cola1, cola2, cola3, elem1, elem2, n1-1, n2-1);
  }
}
```

~~2.4.6.6.2.~~2.3.6.6.2. Ejemplo 2. Realizar la unión de dos colas.

En este ejemplo se parte de dos colas (*cola1* y *cola2*) ordenadas ascendentemente desde el punto de salida hacia el de llegada, y se desea obtener una nueva cola (*cola3*) con todos los elementos de ambas (sin repeticiones), ordenados en el mismo sentido. La Figura 2.12. muestra un ejemplo.

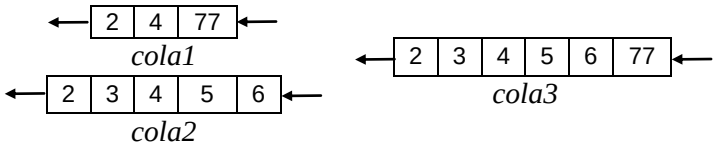

Figura 2.12.Ejemplo de obtención de una cola con los elementos de otras dos

Para resolver este ejemplo, utilizaremos la técnica recursiva conociendo el número de elementos. Como en este caso queremos introducir en la *cola3* todos los elementos de la *cola1* y la *cola2* sin repeticiones, la condición de parada de la recursividad será cuando hayamos tratado todos los elementos de ambas colas (*n1 < 0 && n2 < 0*).

El diseño propuesto requiere el empleo de instrucciones externas al algoritmo recursivo: Previamente es necesario desencolar y encolar de ambas colas para disponer de los valores iniciales de *elem1* y *elem2*.

Así pues la cabecera del método recursivo quedaría:

```
Static void MezclaOrR2 (Cola cola1, Cola cola2,Cola cola3,int elem1,int elem2,int
n1 ,int n2);
```

El tratamiento implica básicamente, lo siguiente:

Fase de ida:

- Comparar los elementos procedentes de ambas colas. Tanto *elem1* como *elem2* proceden de operaciones de desencolar ejecutadas en instancias anteriores por lo que se procederá a encolar el menor de ellos (o uno cualquiera de los dos si son iguales) en *cola3* (como *cola3* debe tener sus elementos ordenados en el mismo sentido que las *cola1* y *cola2*, los elementos se encolan en *cola3* antes de realizar la llamada recursiva).
- Se desencola-encola de las colas pertinentes si es posible (*n1 |2* > 0), pasando así *elem1|2* a la instancia siguiente.

Fase de transición:

- Si una de las colas tiene elementos por tratar y la otra no, se copia el elemento pendiente de tratar en *cola3* y se llama al método auxiliar *copiarcola*, que copia lo que queda de una de las colas en *cola3*.

Fase de "vuelta":

- No hay que realizar ningún tratamiento específico, ni es necesario realizar la operación de encolar dado que ya se ha hecho en la fase de "ida" (mecanismo desencolar-encolar).

A continuación se muestra el algoritmo que resuelve el problema.

```
static void copiarcola (Cola origen, Cola destino, int n) throws ColaVacia {
 int elem;
 if (n > 0) {
    elem = origen.desencolar ();
    destino.encolar (elem);
    origen.encolar (elem);
    copiarcola (origen, destino, n-1);
  }
}
static void mezclaOrR2 (Cola cola1, Cola cola2, Cola cola3, int elem1, int elem2,
int n1 , int n2) throws ColaVacia {
 if ((n1 >= 0) && (n2 >= 0))
    if (elem1 < elem2) {
       cola3.encolar (elem1);
       if (n1 >0) {
          elem1 = cola1.desencolar ();
          cola1.encolar (elem1);
        }
       mezclaOrR2 (cola1, cola2, cola3, elem1, elem2, n1-1, n2);
     }
    else if (elem1 > elem2) {
            cola3.encolar (elem2);
            if (n2 > 0) {
               elem2 = cola2.desencolar ();
               cola2.encolar (elem2);
             }
            mezclaOrR2 (cola1,cola2,cola3,elem1,elem2,n1,n2-1);
          }
         else {
            cola3.encolar (elem1);
            if (n1 > 0) {
               elem1 = cola1.desencolar ();
               cola1.encolar (elem1);
             }
            if (n2 > 0) {
               elem2 = cola2.desencolar ();
               cola2.encolar (elem2);
             }
            mezclaOrR2(cola1,cola2,cola3,elem1,elem2,n1-1,n2-1);
          }
 else {
    if (n1 >= 0) {
       cola3.encolar (elem1);
       copiarcola (cola1, cola3, n1);
     }
    else if (n2 >= 0) {
            cola3.encolar (elem2);
            copiarcola (cola2, cola3, n2);
          }
     }
}
```

```
static void mezclarColaOrR2 (Cola cola1, Cola cola2, Cola cola3) throws ColaVacia {
 int elem1 = -9999, elem2 = -9999, n1 , n2;
 n1 = cola1.numElemCola ();
 n2 = cola2.numElemCola ();
 if (n1 > 0) {
    elem1 = cola1.desencolar ();
    cola1.encolar (elem1);
  }
 if (n2 > 0) {
    elem2 = cola2.desencolar ();
    cola2.encolar (elem2);
  }
 mezclaOrR2 (cola1, cola2, cola3, elem1, elem2, n1-1, n2-1);
}
```

~~2.4.6.7.~~2.3.6.7. Mezcla de colas con terminación anticipada.

Dadas dos colas (*cola1* y *cola2*) ordenadas ascendentemente desde el inicio hasta el final (sin elementos repetidos en cada una de las colas), se desea realizar un método booleano que indique si los elementos de la *cola2* están contenidos en la *cola1*.

Este ejemplo se puede resolver a partir de la consideración de mezcla de colas en la modalidad AND (termina cuando se ha procesado por completo cualquiera de las colas).

En este caso se produce una situación de terminación anticipada cuando, en su caso, aparece por primera vez un elemento de *cola2* que no está en *cola1* (*elem1 > elem2*). El proceso debe terminar y el método devuelve false.

En caso contrario (terminación "pesimista") pueden darse las siguientes circunstancias:

- Se ha procesado completa *cola1* (*n1 < 0*) pero *cola2* no (*n2 >=0*). Aún quedan elementos en *cola2* por tanto no cumple la condición de "estar contenida" en *cola1*, y *en consecuencia* el resultado será *false*.
- Se ha procesado *cola2* completa (*n2 < 0*) y pero *cola1* no (*n1 >= 0*). Por tanto, *cola2* estará contenida en *cola1*, y *en consecuencia* el resultado será *true*.
- Se ha conseguido llegar a la terminación "pesimista" (se han procesado completamente ambas colas: *n1 < 0* y *n2 < 0*). Por lo tanto, *cola2* está contenida en *cola1* y el método devolverá el valor *true*.

El diseño propuesto requiere el empleo de instrucciones externas al algoritmo recursivo: Previamente es necesario desencolar-encolar de ambas colas para disponer de los valores iniciales de *elem1* y *elem2*.

Aplicando a este caso el tratamiento genérico, se trata de:

Fase de "ida".

- Desencolar-encolar si procede (en función del valor de los argumentos *n1* y *n2)*. Se pueden dar los siguientes casos:

- *elem1* > *elem2*. Es imposible que la *cola2* se encuentre contenida en la *cola1*, por lo que habrá que recolocar ambas *colas*, utilizando el método auxiliar *ordenada*, y devolver *false* como resultado.
- *elem1* < *elem2*. Hay que desencolar *de cola1* en la siguiente ejecución. La llamada recursiva se hace con *n1-1* y *n2*.
- *elem1* == *elem.* La llamada recursiva se hace con *n1-1* y *n2-1*.

Fase de "transición"

- Si queda algún elemento pendiente de tratar en alguna de las colas (*n1 >= 0* o *n2 >= 0*), habrá que reordenar la cola que corresponda (con *ordenada*) y asignar el resultado correspondiente al método.

Fase de "vuelta":

- No hay que realizar ningún tratamiento específico, ni es necesario realizar la operación de encolar dado que ya se ha hecho en la fase de "ida" (mecanismo desencolar-encolar).

El código se muestra a continuación:

```
static void ordenar (Cola cola, int n) throws ColaVacia {
 int elem;
 if (n > 0) {
    elem = cola.desencolar ();
    cola.encolar (elem);
    ordenar (cola, n-1);
  }
}
static boolean contenida (Cola cola1, Cola cola2, int elem1, int elem2, int n1 , int
n2) throws ColaVacia {
 boolean resul;
 if ((n1 >= 0) && (n2 >= 0))
    if (elem1 < elem2) {
       if (n1 > 0) {
          elem1 = cola1.desencolar ();
          cola1.encolar (elem1);
        }
       resul = contenida (cola1, cola2, elem1, elem2, n1-1, n2);
     }
    else if (elem1 > elem2) {
             ordenar (cola1, n1 );
             ordenar (cola2, n2);
             resul = false;
           }
          else {
             if (n1 > 0) {
                elem1 = cola1.desencolar ();
                cola1.encolar (elem1);
              }
             if (n2 > 0) {
                elem2 = cola2.desencolar ();
                cola2.encolar (elem2);
              }
             resul = contenida (cola1, cola2, elem1, elem2,n1-1,n2-1);
           }
 else {
    if (n1 >= 0) {
       ordenar (cola1, n1);
       resul = true;
     }
```

```
    else if (n2 >= 0) {
            ordenar (cola2, n2);
            resul = false;
          }
        else resul = true;
  }
 return resul;
}
static boolean estaContenida (Cola cola1, Cola cola2) throws ColaVacia {
 int elem1, elem2, n1, n2;
 boolean resul;

 n1 = cola1.numElemCola ();
 n2 = cola2.numElemCola ();
 if ((n1 > 0) && (n2 > 0)) {
    elem1 = cola1.desencolar ();
    cola1.encolar (elem1);
    elem2 = cola2.desencolar ();
    cola2.encolar (elem2);
    resul = contenida (cola1, cola2, elem1, elem2, n1-1, n2-1);
  }
 else if (n2 > 0)
        resul = false;
      else resul = true;
 return resul;
}
```

~~2.5.~~2.4. CONSTRUCCIÓN DE UN TAD A PARTIR DE OTRO.

Se pretende en este apartado mostrar la posibilidad de construir nuevos TAD a partir de otros. A continuación se muestra un ejemplo en el que se construye a partir del TAD pila un TAD vector de números enteros tad_Vector. Dispone de las siguientes funcionalidades:

- **crearVector** (). Prepara un nuevo vector de tipo TSVector. El método solicita al usuario el número de elementos y los inicializa con 0.

- **escribirVector** (*pos*, *dato*). Método que lleva a la posición *pos* del *vector* el valor indicado en el argumento *dato*. Deberá de verificar que se trata de una posición válida (no superior al número de elementos indicado en la creación del vector).

- **leerVector** (*pos*). Método que devuelve el contenido de la posición *pos* del *vector*. Deberá verificar, igualmente, que se trata de una posición válida.

- **tamaño** ().Método que devuelve el tamaño actual del vector.

En virtud de la característica de ocultamiento, propia de la los TAD's, el usuario imagina y trabaja con algo que se comporta como un vector sin saber cómo está construido (Figura 2.13.).

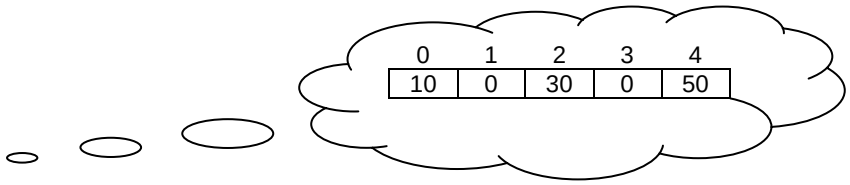

Figura 2.13. TAD Vector.

Primero definiríamos un interfaz en el que indicaríamos cuales son las operaciones del nuevo TAD:

```
import java.io.IOException;
public interface Vector {
   void crearVector () throws NumberFormatException, IOException;
   int leerVector (int posicion) throws PilaVacia;
   void escribirVector (int posicion, int contenido) throws PilaVacia;
   int tamaño ();
}
```

A continuación, se construyen las operaciones. Se utilizan dos métodos auxiliares (leer y escribir), que se definen como privados.

```
import java.io.IOException;
import java.io.InputStreamReader;

import tadPila.Pila;
import tadPila.TadPila;
import tadPila.PilaVacia;

public class TadVector implements Vector {
   Pila vector;
   public int tamano;

   public void crearVector () throws NumberFormatException, IOException {
      int  i;
      BufferedReader linea = new BufferedReader(new InputStreamReader
(System.in));
      Pila auxv = new TadPila ();

      System.out.println ("Numero de elementos: ");
      tamano = Integer.parseInt(linea.readLine());
      for (i = 1; i <= tamano; i++)
        auxv.apilar (0);
      vector = auxv;
   }
   private int leer (int i, int posicion, Pila vector) throws PilaVacia {
      int  resul, elem;

      if (i < posicion) {
        elem = vector.desapilar ();
        resul = leer (i+1, posicion, vector);
        vector.apilar (elem);
       }
      else {
        elem = vector.desapilar ();
        resul = elem;
        vector.apilar (elem);
       }
      return resul;
   }
   public int leerVector (int posicion) throws PilaVacia {
      int  resul, i;

      if ((posicion > tamano) || (posicion <= 0)) {
        System.out.println ("Error. Valor fuera de rango");
        resul = 99999;
       }
      else {
        i = 0;
        resul = leer (i+1, posicion, vector);
       }
      return resul;
   }
   private void escribir (int i, int posicion, int contenido, Pila vector) throws
PilaVacia {
      int elem;

      if (i < posicion) {
        elem = vector.desapilar ();
        escribir (i+1, posicion, contenido, vector);
        vector.apilar (elem);
      }
      else {
        elem = vector.desapilar ();
        vector.apilar (contenido);
      }
   }
```

```
   public void escribirVector (int posicion, int contenido) throws PilaVacia {
      if ((posicion > tamano) || (posicion < 0))
         System.out.println ("Error. Indice fuera de rango");
      else
         escribir (1, posicion, contenido, vector);
   }
   public int tamaño () {
      return tamano;
   }
}
```

El siguiente es un sencillo programa que prueba el nuevo TAD.

```
import java.io.IOException;

public class PruebaVector {
   public static void main(String[] args) throws NumberFormatException,
IOException, Pilavacia {
      Vector v = new TadVector ();
      int d,i;

      v.crearVector ();
      v.escribirVector (5, 77);
      v.escribirVector (3, 87);
      v.escribirVector (8, 97);
      for (i = 1; i <= v.tamaño (); i++) {
         d = v.leerVector (i);
         System.out.println ("El dato de la posicion "+i+" es: "+d);
      }
   }
}
```